МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра №806 «Вычислительна математика и программирование»

Курсовая работа по курсу “Дискретный анализ”

Персистентные структуры данных

Студент: Караев Тариел Жоомартбекович

Группа: М8О-308Б-22

Преподаватель: Макаров Никита Константинович

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2024

Содержание

Репозиторий

`https://github.com/SempaiTakoo/DA_labs`

Постановка задачи

Программа должна читать входные данные из стандартного потока ввода и выводить ответ на стандартный поток вывода. Реализуйте алгоритм $A\star$ для неориентированного графа. Расстояние между соседями вычисляется как простое евклидово расстояние на плоскости.

Формат ввода

В первой строке вам даны два числа n и m ($1 \leq n \leq 10^4, 1 \leq m \leq 10^5$) вершин и рёбер в графе. В следующих n строках вам даны пары чисел , $(-10^9 \leq x, y \leq 10^9)$, описывающие положение вершин графа в двумерном пространстве. В следующих m строках даны пары чисел в отрезке от 1 до n, описывающие рёбра графа. Далее дано число q ($1 \leq q \leq 300$) и в следующих q строках даны запросы в виде пар чисел a, b ($1 \leq , b \leq n$) на поиск кратчайшего пути между двумя вершинами.

Формат вывода

В ответ на каждый запрос выведите единственное число — длину кратчайшего пути между заданными вершинами с абсолютной либо относительной точностью 10^{-6}, если пути между вершинами не существует выведите -1.

Метод решения

Как сказано в условии задачи, необходимо реализовать алгоритм $A\star$, позволяющий находить кратчайшие пути между вершинами графа. В качестве эвристической функции используется евклидово расстояние на плоскости.

В процессе работы алгоритм поддерживает две структуры: множество "открытых"вершин, которые ещё предстоит обработать, и множество "закрытых"вершин, которые уже обработаны. Пока множество "открытых"вершин непустое, из него извлекается вершина с наименьшим значением оценки f, после чего анализируются её соседи. Если новая оценка пути до соседней вершины оказывается меньше ранее известной, оценка пути обновляется и вершина добавляется во множество "открытых"вершин.

Применение эвристики позволяет алгоритму $A\star$ быстрее находить оптимальный маршрут, избегая лишних обходов, что положительно выделяет его на фоне алгоритма Дийкстры, чьим обобщением он является.

Формально, алгоритм $A\star$ имеет сложность $O((n+m)\log n$, так как при использовании очереди с приоритетом требуется $n+m$ раз вставить и извлечь вершину графа со сложностью $\log n$. Однако благодаря эвристике реальная скорость $A\star$ близится к $O(n)$.

Описание алгоритма

Основные этапы работы алгоритма:

- Инициализация:
  1. Создаются множества "открытых"и "закрытых"вершин.
  2. Для начальной вершины устанавливается нулевая стоимость пути, а её эвристическая оценка рассчитывается на основе евклидова расстояния до конечной вершины.
- Основной цикл поиска:
  1. Из "открытого"множества выбирается вершина с наименьшим значением функции оценки.
  2. Если это целевая вершина, алгоритм завершается, возвращая найденную стоимость пути.
  3. Вершина переносится в "закрытое"множество, после чего анализируются все её соседние вершины.
- Обновление данных соседних вершин:
  1. Если сосед уже находится в "закрытом"множестве, он пропускается.
  2. Для каждой соседней вершины пересчитывается стоимость пути, и если найден более короткий маршрут, обновляются соответствующие данные.
  3. Соседняя вершина добавляется в "открытое"множество и её приоритет обновляется.

В случае отсутствия пути между вершинами алгоритм $A\star$ корректно возвращает отрицательное значение, что означает недостижимость цели.

Описание программы

- **Graph** – класс, представляющий граф. Содержит список вершин с координатами и список смежности для рёбер.
  - **addVertex** – метод, добавляющий вершину с заданными координатами.
  - **addEdge** – метод, создающий ребро между двумя вершинами, дополнительно вычисляя расстояние между ними.
- **Node** – вспомогательная структура для приоритетной очереди в алгоритме $A\star$, содержащая номер вершины и оценку стоимости пути f.
- **get_distance** – вычисляет евклидово расстояние между двумя точками.
- **main** – точка входа в программу, отвечает за инициализацию и запуск основного процесса вычислений.
- **input_graph** – функция для считывания данных о графе из входного потока.
- **shortest_paths_between_vertices** – управляет процессом поиска кратчайших путей между заданными парами вершин, используя алгоритм $A\star$.
- **a_star** – основная функция реализации алгоритма $A\star$, осуществляет поиск кратчайшего пути в графе.

Исходныйкод

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <limits>
#include <set>
#include <cmath>
#include <set>
#include <iomanip>

using namespace std;

const bool DEBUG = false;
const double INF = numeric_limits<double>::infinity();

double get_distance(pair<int, int> a, pair<int, int> b) {
    long x = b.first - a.first;
    long y = b.second - a.second;
    return sqrt(x * x + y * y);
}
```

```
class Graph {
public:
    vector<pair<double, double>> vertices;
    vector<vector<pair<int, double>>> adjacency_list;

    Graph(int vertices_count) {
        vertices.resize(vertices_count);
        adjacency_list.resize(vertices_count);
    }

    void addVertex(int vertex_number, double x, double y) {
        vertices[vertex_number] = {x, y};
    }

    void addEdge(int s, int f) {
        double distance = get_distance(vertices[s], vertices[f]);
        adjacency_list[s].push_back({f, distance});
        adjacency_list[f].push_back({s, distance});
    }
};

struct Node {
    int vertex_number;
    double f;

    bool operator>(const Node node) const {
        return f > node.f;
    }
};

Graph input_graph() {
    int n, m;
    cin >> n >> m;

    Graph graph(n);

    int x, y;
    for (int i = 0; i < n; ++i)
    {
        cin >> x >> y;
        graph.addVertex(i, x, y);
    }

    int s, f;
    for (int i = 0; i < m; ++i)
    {
        cin >> s >> f;
        graph.addEdge(s - 1, f - 1);
    }

    return graph;
```

```
}

double a_star(Graph& graph, int start, int finish) {
    size_t dim = graph.vertices.size();

    priority_queue<Node, vector<Node>, greater<Node>> open;
    vector<bool> closed(dim, false);
    vector<double> g(dim, INF);
    vector<double> f(dim, INF);

    g[start] = 0;
    f[start] = get_distance(graph.vertices[start], graph.vertices[finish])
       ;
    open.push({start, f[start]});

    while (!open.empty()) {
        Node node = open.top();
        int current = node.vertex_number;
        open.pop();
        closed[current] = true;

        if (current == finish) {
            return g[current];
        }

        for (pair<int, double> edge : graph.adjacency_list[current]) {
            int neighbor = edge.first;
            double distance = edge.second;

            if (closed[neighbor]) {
                continue;
            }

            double g_temp = g[current] + distance;
            if (g_temp >= g[neighbor]) {
                continue;
            }

            g[neighbor] = g_temp;
            f[neighbor] = g[neighbor] + get_distance(
                graph.vertices[neighbor],
                graph.vertices[finish]
            );
            open.push({neighbor, f[neighbor]});
        }
    }

    return -1;
}

void shortest_paths_between_vertices() {
```

```
    Graph graph = input_graph();

    int q;
    cin >> q;

    for (int i = 0; i < q; ++i) {
        int start, finish;
        cin >> start >> finish;

        --start;
        --finish;

        double answer = a_star(graph, start, finish);

        if (answer == -1) {
            cout << -1 << '\n';
        } else {
            cout << fixed << setprecision(6) << answer << '\n';
        }
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    shortest_paths_between_vertices();

    return 0;
}
```

Тест производительности

В данной главе представлены результаты тестирования производительности алгоритма $A\star$ на случайных графах. Были проведены тесты при зафиксированных $m = 2n$, $q = 1$ и изменении $n = \alpha \cdot 10^4$, $\alpha = 1, 2, ..., 10$. Данные приведены в таблице 1.

n	$(n+m)\log(n)$	Время выполнения 10^{-7} с
10000	120000	28866,6
20000	258061,7997	50367,6
30000	402940,9129	54438,6
40000	552247,199	26760,4
50000	704845,5007	86821,7
60000	860067,2251	199223
70000	1017470,588	153932
80000	1176741,597	80430,6
90000	1337645,478	508806
100000	1500000	82967,5

Таблица 1: Результаты тестирования производительности

Далее на рисунке 1 изображены два графика:

- график функции $(n+m)\log(n)$, являющийся асимптотической верхней границей сложности алгоритма $A\star$,
- график зависимости времени выполнения программы от размера входных данных.

Как видно по графикам, благодаря представлению вершин графа в виде координат и эвристике, выражающейся в евклидовом расстоянии на плоскости, алгоритм работает гораздо быстрее своей асимптотической верхней границы $(n+m)\log(n)$. Это обусловлено тем, что эвристика позволяет делать меньше лишних обходов вершин.

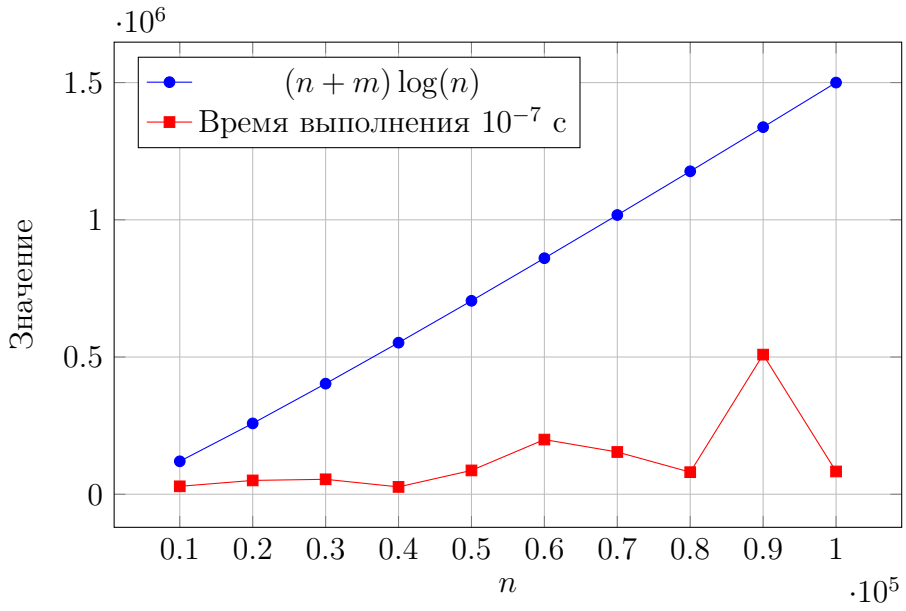

Рис. 1: График зависимости $(n+m)\log(n)$ и времени выполнения от n

Выводы

В ходе выполнения данной курсовой работы был реализован алгоритм $A\star$ для поиска кратчайшего пути между вершинами графа. Было изучено, что в определённых задачах дополнительная информация в виде эвристики позволяет существенно улучшить эффективность алгоритма.

Также в ходе работы были получены навыки оптимизации программ на C++. В частности, более вдумчивый подход при выборе различных типов и структур данных, которые могут значительно снизить издержки по памяти и ускорить работу программы.